Data Transformation

Contents

1 Review **1**

2 Introduction **1**

3 Functions for Rows **5**
- 3.1 `filter()` . . . 5
- 3.2 Common mistakes . . . 7
- 3.3 `arrange()` . . . 7

4 Functions for Columns **9**
- 4.1 `mutate()` . . . 9
- 4.2 `select()` . . . 11
- 4.3 The `$` . . . 13
- 4.4 `rename()` . . . 14
- 4.5 `relocate()` . . . 14

5 Creating Groups **15**
- 5.1 `group_by()` . . . 16
- 5.2 `summarize()` . . . 16
- 5.3 Grouping by multiple variables . . . 17
- 5.4 Ungrouping . . . 19

1 Review

So far we have:

- learned how to load in packages
- learned how to create some plots with built in datasets
- Some basics of coding
- Basics of naming objects
- General principals of calling functions.

This week we will learn:

- create new variables
- summarize existing variables
- reorder datasets

2 Introduction

In this chapter we'll focus on the `dplyr` package which is part of the `tidyverse`. This package has really transformed the way people use `R`. To the point where you can tell if someone learned `R` before or after `dplyr` was developed.

To illustrate the methods we'll use the `nycflights13` package, and use `ggplot2` to help us understand the data.

If you don't have the `nycflights13` package, you will have to install it using `install.packages("nycflights13")`.

```
library(nycflights13)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.8     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Note the conflict messages that's printed when you load the tidyverse.

It tells you that dplyr overwrites some functions in base R. If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

So far we've mostly ignored which package a function comes from because most of the time it doesn't matter. When it does you'll need to use `packagename::functionname()`.

nycflights13

To explore the basic dplyr verbs, we're going to use `flights` dataset which is contained in `nycflights13`. This dataset contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?flights`.

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

The flights data set is a **tibble** – a special type of data frame used by the tidyverse to avoid some common gotchas.

The most important difference is the way it prints: tibbles are designed for large datasets, so they only show the first few rows and only the columns that fit on one screen.

To see everything, use `View(flights)` to open the dataset in the RStudio viewer.

Quick detour on variable types

Below are examples of data types in R:

- character: "a", "swc"
- numeric: 2, 15.5
- integer: 2L (the L tells R to store this as an integer)
- logical: TRUE, FALSE
- factors: categorical data (e.g., 1/2/3, or A/B/C, or "male"/"female")
- complex: 1+4i (complex numbers with real and imaginary parts)

Note that each variable type is included when the data are printed above. This contains: `<int>` is short for integer, `<dbl>` is short for double (aka numeric, real numbers), `<chr>` for character (aka strings), and `<dttm>` for date-time.

dplyr basics

Below were going explore doing using `dplyr` for doing some data manipulations. Here are some initial points about using dplyr:

1. The first argument is always a data frame (not a matrix).
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is always a new data frame.

pipes

Because the first argument is a data frame and the output is a data frame, dplyr verbs work work well with the pipes, `|>`.

The pipe takes the thing on its left and passes it along to the function on its right:

```
+ `x |> f(y)` is equivalent to `f(x, y)`, and
+ `x |> f(y) |> g(z)` is equivalent to into `g(f(x, y), z)`.
```

The easiest way to pronounce the pipe is "then". That makes it possible to get a sense of the following code even though you haven't yet learned the details:

```
flights |>
  filter(dest == "IAH") |>
  group_by(year, month) |>
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

```
## `summarise()` has grouped output by 'year'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##     year month arr_delay
##    <int> <int>     <dbl>
##  1  2013     1     4.16
##  2  2013     2     5.40
##  3  2013     3    -1.19
##  4  2013     4    14.8
##  5  2013     5     0.972
##  6  2013     6    11.1
##  7  2013     7    11
##  8  2013     8     0.705
##  9  2013     9   -10.6
```

```
## 10  2013    10     1.81
## 11  2013    11    -1.78
## 12  2013    12    14.5
```

What happens in the code? The code starts with the flights dataset, then filters it so only a specific destination is used ("IAH"), then groups it by month/year, then summarizes it. We'll go through these quantities as we go along.

It's important to know that `%>%` can also be used for pipes. In fact most of the help files you'll see will use `%>%` instead of `|>`.

```
flights %>%
  filter(dest == "IAH") %>%
  group_by(year, month) %>%
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

```
## `summarise()` has grouped output by 'year'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##     year month arr_delay
##    <int> <int>     <dbl>
##  1  2013     1     4.16
##  2  2013     2     5.40
##  3  2013     3    -1.19
##  4  2013     4    14.8
##  5  2013     5     0.972
##  6  2013     6    11.1
##  7  2013     7    11
##  8  2013     8     0.705
##  9  2013     9   -10.6
## 10  2013    10     1.81
## 11  2013    11    -1.78
## 12  2013    12    14.5
```

I don't care which you use.

If you didn't use pipes, you could run the above code as follows:

```
filt_flights <-   filter(flights, dest == "IAH")
grp_filt_flights <-   group_by(filt_flights,
                               year, month)
  summarize(grp_filt_flights,
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

```
## `summarise()` has grouped output by 'year'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##     year month arr_delay
##    <int> <int>     <dbl>
##  1  2013     1     4.16
##  2  2013     2     5.40
##  3  2013     3    -1.19
```

```
##  4  2013     4    14.8
##  5  2013     5     0.972
##  6  2013     6    11.1
##  7  2013     7    11
##  8  2013     8     0.705
##  9  2013     9   -10.6
## 10  2013    10     1.81
## 11  2013    11    -1.78
## 12  2013    12    14.5
```

Pipes make the code look cleaner and you don't have create extra objects.

dplyr's verbs (functions) are organised into four groups based on what they operate on: **rows**, **columns**, **groups**, or **tables**.

In the following sections you'll learn the most important verbs for rows, columns, and groups. We'll come back to tables later.

3 Functions for Rows

The most important verbs that operate on rows are `filter()`, which changes which rows are present without changing their order, and `arrange()`, which changes the order of the rows without changing which are present. Both functions only affect the rows, and the columns are left unchanged.

3.1 `filter()`

`filter()` allows you to keep rows based on the values of the columns[1]. The first argument is the data frame. The second and subsequent arguments are the conditions that must be true to keep the row. For example, we could find all flights that arrived more than 120 minutes (two hours) late:

```
flights |>
  filter(arr_delay > 120)
```

```
## # A tibble: 10,034 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      811        630     101    1047     830     137 MQ
##  2  2013     1     1      848       1835     853    1001    1950     851 MQ
##  3  2013     1     1      957        733     144    1056     853     123 UA
##  4  2013     1     1     1114        900     134    1447    1222     145 UA
##  5  2013     1     1     1505       1310     115    1638    1431     127 EV
##  6  2013     1     1     1525       1340     105    1831    1626     125 B6
##  7  2013     1     1     1549       1445      64    1912    1656     136 EV
##  8  2013     1     1     1558       1359     119    1718    1515     123 EV
##  9  2013     1     1     1732       1630      62    2028    1825     123 EV
## 10  2013     1     1     1803       1620     103    2008    1750     138 MQ
## # ... with 10,024 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

The logical arguments that you can use are:

[1]Later, you'll learn about the `slice_*()` family which allows you to choose rows based on their positions

- `>` (greater than),
- `>=` (greater than or equal to),
- `<` (less than),
- `<=` (less than or equal to),
- `==` (equal to), and
- `!=` (not equal to).

You can combine multiple logical phrases using `&` (and) or `|` (or) to combine multiple conditions:

```
# Flights that departed on January 1
flights |>
  filter(month == 1 & day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 832 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

```
# Flights that departed in January or February
flights |>
  filter(month == 1 | month == 2)
```

```
## # A tibble: 51,955 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 51,945 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

There's a useful shortcut when you're combining `|` and `==`: `%in%`. It keeps rows where the variable equals one of the values on the right:

```
# A shorter way to select flights that departed in January or February
flights |>
  filter(month %in% c(1, 2))
```

```
## # A tibble: 51,955 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 51,945 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

We can use these to create a new data frame that may be used later in plotting or analyses:

```
Jan_Feb_flights <- flights |>
  filter(month %in% c(1, 2))
```

3.2 Common mistakes

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. `filter()` will let you know when this happens:

```
flights |>
  filter(month = 1)
```

```
## Error in `filter()`:
## ! We detected a named input.
## i This usually means that you've used `=` instead of `==`.
## i Did you mean `month == 1`?
```

Another mistakes is you write "or" statements like you would in English:

```
flights |>
  filter(month == 1 | 2)
```

3.3 `arrange()`

`arrange()` changes the order of the rows based on the value of the columns. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns. For example, the following code sorts by the departure time, which is spread over four columns.

```
flights |>
  arrange(year, month, day, dep_time)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

You can use `desc()` to re-order by a column in descending order. For example, this code shows the most delayed flights:

```
flights |>
  arrange(desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     9      641        900    1301    1242    1530    1272 HA
##  2  2013     6    15     1432       1935    1137    1607    2120    1127 MQ
##  3  2013     1    10     1121       1635    1126    1239    1810    1109 MQ
##  4  2013     9    20     1139       1845    1014    1457    2210    1007 AA
##  5  2013     7    22      845       1600    1005    1044    1815     989 MQ
##  6  2013     4    10     1100       1900     960    1342    2211     931 DL
##  7  2013     3    17     2321        810     911     135    1020     915 DL
##  8  2013     6    27      959       1900     899    1236    2226     850 DL
##  9  2013     7    22     2257        759     898     121    1026     895 DL
## 10  2013    12     5      756       1700     896    1058    2020     878 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

You can combine `arrange()` and `filter()` to solve more complex problems. For example, we could look for the flights that were most delayed on arrival that left on roughly on time:

```
flights |>
  filter(dep_delay <= 10 & dep_delay >= -10) |>
  arrange(desc(arr_delay))
```

```
## # A tibble: 239,109 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013    11     1      658        700      -2    1329    1015     194 VX
##  2  2013     4    18      558        600      -2    1149     850     179 AA
##  3  2013     7     7     1659       1700      -1    2050    1823     147 US
##  4  2013     7    22     1606       1615      -9    2056    1831     145 DL
##  5  2013     9    19      648        641       7    1035     810     145 UA
##  6  2013     4    18      655        700      -5    1213     950     143 AA
##  7  2013     6    30     1423       1425      -2    1816    1554     142 B6
##  8  2013     6    24     1523       1520       3    1931    1710     141 AA
##  9  2013     3    18     1844       1847      -3      39    2219     140 UA
## 10  2013     7     1      905        905       0    1443    1223     140 DL
## # ... with 239,099 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

4 Functions for Columns

There are four important verbs that affect the columns without changing the rows: `mutate()`, `select()`, `rename()`, and `relocate()`.

`mutate()` creates new columns that are functions of the existing columns; `select()`, `rename()`, and `relocate()` change which columns are present, their names, or their positions.

4.1 `mutate()`

The job of `mutate()` is to add new columns that are calculated from the existing columns. Later, we'll discuss a large set of functions that you can use to manipulate different types of variables. For now, we'll stick with basic algebra, which allows us to compute the `gain`, how much time a delayed flight made up in the air, and the `speed` in miles per hour:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 11 more variables: flight <int>, tailnum <chr>,
```

```
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, gain <dbl>, speed <dbl>, and abbreviated
## #   variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
## #   4: sched_arr_time, 5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

By default, `mutate()` adds new columns on the right hand side of your dataset, which makes it difficult to see what's happening here.

We can use the `.before` argument to instead add the variables to the left hand side[2]:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

```
## # A tibble: 336,776 x 21
##     gain speed  year month   day dep_t~1 sched~2 dep_d~3 arr_t~4 sched~5 arr_d~6
##    <dbl> <dbl> <int> <int> <int>   <int>   <int>   <dbl>   <int>   <int>   <dbl>
##  1    -9  370.  2013     1     1     517     515       2     830     819      11
##  2   -16  374.  2013     1     1     533     529       4     850     830      20
##  3   -31  408.  2013     1     1     542     540       2     923     850      33
##  4    17  517.  2013     1     1     544     545      -1    1004    1022     -18
##  5    19  394.  2013     1     1     554     600      -6     812     837     -25
##  6   -16  288.  2013     1     1     554     558      -4     740     728      12
##  7   -24  404.  2013     1     1     555     600      -5     913     854      19
##  8    11  259.  2013     1     1     557     600      -3     709     723     -14
##  9     5  405.  2013     1     1     557     600      -3     838     846      -8
## 10   -10  319.  2013     1     1     558     600      -2     753     745       8
## # ... with 336,766 more rows, 10 more variables: carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: dep_time, 2: sched_dep_time, 3: dep_delay, 4: arr_time,
## #   5: sched_arr_time, 6: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

The `.` is a sign that `.before` is an argument to the function, not the name of a new variable. You can also use `.after` to add after a variable, and in both `.before` and `.after` you can the name of a variable name instead of a position. For example, we could add the new variables after `day`:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .after = day
  )
```

```
## # A tibble: 336,776 x 21
##     year month   day  gain speed dep_t~1 sched~2 dep_d~3 arr_t~4 sched~5 arr_d~6
##    <int> <int> <int> <dbl> <dbl>   <int>   <int>   <dbl>   <int>   <int>   <dbl>
##  1  2013     1     1    -9  370.     517     515       2     830     819      11
##  2  2013     1     1   -16  374.     533     529       4     850     830      20
##  3  2013     1     1   -31  408.     542     540       2     923     850      33
##  4  2013     1     1    17  517.     544     545      -1    1004    1022     -18
```

[2]Remember that in RStudio, the easiest way to see a dataset with many columns is `View()`.

```
##  5  2013     1     1    19  394.     554     600      -6     812     837     -25
##  6  2013     1     1   -16  288.     554     558      -4     740     728      12
##  7  2013     1     1   -24  404.     555     600      -5     913     854      19
##  8  2013     1     1    11  259.     557     600      -3     709     723     -14
##  9  2013     1     1     5  405.     557     600      -3     838     846      -8
## 10  2013     1     1   -10  319.     558     600      -2     753     745       8
## # ... with 336,766 more rows, 10 more variables: carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: dep_time, 2: sched_dep_time, 3: dep_delay, 4: arr_time,
## #   5: sched_arr_time, 6: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Alternatively, you can control which variables are kept with the `.keep` argument. A particularly useful argument is `"used"` which allows you to see the inputs and outputs from your calculations:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

```
## # A tibble: 336,776 x 6
##    dep_delay arr_delay air_time  gain hours gain_per_hour
##        <dbl>     <dbl>    <dbl> <dbl> <dbl>         <dbl>
##  1         2        11      227    -9 3.78          -2.38
##  2         4        20      227   -16 3.78          -4.23
##  3         2        33      160   -31 2.67         -11.6
##  4        -1       -18      183    17 3.05           5.57
##  5        -6       -25      116    19 1.93           9.83
##  6        -4        12      150   -16 2.5           -6.4
##  7        -5        19      158   -24 2.63          -9.11
##  8        -3       -14       53    11 0.883         12.5
##  9        -3        -8      140     5 2.33           2.14
## 10        -2         8      138   -10 2.3           -4.35
## # ... with 336,766 more rows
## # i Use `print(n = ...)` to see more rows
```

4.2 `select()`

It's not uncommon to get datasets with hundreds or even thousands of variables. In this situation, the first challenge is often just focusing on the variables you're interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables. `select()` is not terribly useful with the flights data because we only have 19 variables, but you can still get the general idea of how it works:

```
# Select columns by name
flights |>
  select(year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
```

```
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
## # i Use `print(n = ...)` to see more rows
```

```
# Select all columns between year and day (inclusive)
flights |>
  select(year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
## # i Use `print(n = ...)` to see more rows
```

```
# Select all columns except those from year to day (inclusive)
flights |>
  select(!year:day)
```

```
## # A tibble: 336,776 x 16
##    dep_t~1 sched~2 dep_d~3 arr_t~4 sched~5 arr_d~6 carrier flight tailnum origin
##      <int>   <int>   <dbl>   <int>   <int>   <dbl> <chr>    <int> <chr>   <chr>
##  1     517     515       2     830     819      11 UA        1545 N14228  EWR
##  2     533     529       4     850     830      20 UA        1714 N24211  LGA
##  3     542     540       2     923     850      33 AA        1141 N619AA  JFK
##  4     544     545      -1    1004    1022     -18 B6         725 N804JB  JFK
##  5     554     600      -6     812     837     -25 DL         461 N668DN  LGA
##  6     554     558      -4     740     728      12 UA        1696 N39463  EWR
##  7     555     600      -5     913     854      19 B6         507 N516JB  EWR
##  8     557     600      -3     709     723     -14 EV        5708 N829AS  LGA
##  9     557     600      -3     838     846      -8 B6          79 N593JB  JFK
## 10     558     600      -2     753     745       8 AA         301 N3ALAA  LGA
## # ... with 336,766 more rows, 6 more variables: dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>, and abbreviated
## #   variable names 1: dep_time, 2: sched_dep_time, 3: dep_delay, 4: arr_time,
## #   5: sched_arr_time, 6: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

```
# Select all columns that are characters
flights |>
```

```
  select(where(is.character))
```

```
## # A tibble: 336,776 x 4
##    carrier tailnum origin dest
##    <chr>   <chr>   <chr>  <chr>
##  1 UA      N14228  EWR    IAH
##  2 UA      N24211  LGA    IAH
##  3 AA      N619AA  JFK    MIA
##  4 B6      N804JB  JFK    BQN
##  5 DL      N668DN  LGA    ATL
##  6 UA      N39463  EWR    ORD
##  7 B6      N516JB  EWR    FLL
##  8 EV      N829AS  LGA    IAD
##  9 B6      N593JB  JFK    MCO
## 10 AA      N3ALAA  LGA    ORD
## # ... with 336,766 more rows
## # i Use `print(n = ...)` to see more rows
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with "abc" (this can be *very* useful).
- `ends_with("xyz")`: matches names that end with "xyz".
- `contains("ijk")`: matches names that contain "ijk".
- `num_range("x", 1:3)`: matches `x1`, `x2` and `x3`.

See `?select` for more details.

You can rename variables as you `select()` them by using `=`. The new name appears on the left hand side of the `=`, and the old variable appears on the right hand side:

```
flights |>
  select(tail_num = tailnum) |>
  print(n=6)
```

```
## # A tibble: 336,776 x 1
##   tail_num
##   <chr>
## 1 N14228
## 2 N24211
## 3 N619AA
## 4 N804JB
## 5 N668DN
## 6 N39463
## # ... with 336,770 more rows
## # i Use `print(n = ...)` to see more rows
```

4.3 The $

This is not a function in dplyr, but a basic way to look at a variable in `R`. I'll use it here, because it is commonly what I will use:

```
flights$tailnum
```

I'm not going to evaluate this, because it would show all $> 300,000$ values of the `tailnum`. To only show the first 6 values I'll use the `head()` function:

```
head(flights$tailnum)
```

```
## [1] "N14228" "N24211" "N619AA" "N804JB" "N668DN" "N39463"
```

4.4 rename()

If you just want to keep all the existing variables and just want to rename a few, you can use `rename()` instead of `select()`:

```
flights |>
  rename(tail_num = tailnum)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tail_num <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

It works exactly the same way as `select()`, but keeps all the variables that aren't explicitly selected.

4.5 relocate()

You can move variables around with `relocate()`. By default it moves variables to the front:

```
flights |>
  relocate(time_hour, air_time)
```

```
## # A tibble: 336,776 x 19
##    time_hour           air_t~1  year month   day dep_t~2 sched~3 dep_d~4 arr_t~5
##    <dttm>                <dbl> <int> <int> <int>   <int>   <int>   <dbl>   <int>
##  1 2013-01-01 05:00:00     227  2013     1     1     517     515       2     830
##  2 2013-01-01 05:00:00     227  2013     1     1     533     529       4     850
##  3 2013-01-01 05:00:00     160  2013     1     1     542     540       2     923
##  4 2013-01-01 05:00:00     183  2013     1     1     544     545      -1    1004
##  5 2013-01-01 06:00:00     116  2013     1     1     554     600      -6     812
##  6 2013-01-01 05:00:00     150  2013     1     1     554     558      -4     740
##  7 2013-01-01 06:00:00     158  2013     1     1     555     600      -5     913
##  8 2013-01-01 06:00:00      53  2013     1     1     557     600      -3     709
##  9 2013-01-01 06:00:00     140  2013     1     1     557     600      -3     838
## 10 2013-01-01 06:00:00     138  2013     1     1     558     600      -2     753
## # ... with 336,766 more rows, 10 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   dest <chr>, distance <dbl>, hour <dbl>, minute <dbl>, and abbreviated
## #   variable names 1: air_time, 2: dep_time, 3: sched_dep_time, 4: dep_delay,
```

```
## #   5: arr_time
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

But you can use the same `.before` and `.after` arguments as `mutate()` to choose where to put them:

```
flights |>
  relocate(year:dep_time, .after = time_hour)
```

```
## # A tibble: 336,776 x 19
##    sched_d~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier flight tailnum origin dest
##        <int>   <dbl>   <int>   <int>   <dbl> <chr>    <int> <chr>   <chr>  <chr>
##  1       515       2     830     819      11 UA        1545 N14228  EWR    IAH
##  2       529       4     850     830      20 UA        1714 N24211  LGA    IAH
##  3       540       2     923     850      33 AA        1141 N619AA  JFK    MIA
##  4       545      -1    1004    1022     -18 B6         725 N804JB  JFK    BQN
##  5       600      -6     812     837     -25 DL         461 N668DN  LGA    ATL
##  6       558      -4     740     728      12 UA        1696 N39463  EWR    ORD
##  7       600      -5     913     854      19 B6         507 N516JB  EWR    FLL
##  8       600      -3     709     723     -14 EV        5708 N829AS  LGA    IAD
##  9       600      -3     838     846      -8 B6          79 N593JB  JFK    MCO
## 10       600      -2     753     745       8 AA         301 N3ALAA  LGA    ORD
## # ... with 336,766 more rows, 9 more variables: air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, year <int>, month <int>,
## #   day <int>, dep_time <int>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

```
flights |>
  relocate(starts_with("arr"), .before = dep_time)
```

```
## # A tibble: 336,776 x 19
##     year month   day arr_time arr_delay dep_time sched~1 dep_d~2 sched~3 carrier
##    <int> <int> <int>    <int>     <dbl>    <int>   <int>   <dbl>   <int> <chr>
##  1  2013     1     1      830        11      517     515       2     819 UA
##  2  2013     1     1      850        20      533     529       4     830 UA
##  3  2013     1     1      923        33      542     540       2     850 AA
##  4  2013     1     1     1004       -18      544     545      -1    1022 B6
##  5  2013     1     1      812       -25      554     600      -6     837 DL
##  6  2013     1     1      740        12      554     558      -4     728 UA
##  7  2013     1     1      913        19      555     600      -5     854 B6
##  8  2013     1     1      709       -14      557     600      -3     723 EV
##  9  2013     1     1      838        -8      557     600      -3     846 B6
## 10  2013     1     1      753         8      558     600      -2     745 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: sched_arr_time
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

5 Creating Groups

In this section, we'll focus on the most important functions: `group_by()`, `summarize()`, and the slice family of functions.

5.1 group_by()

Use `group_by()` to divide your dataset into groups meaningful for your analysis:

```
flights |>
  group_by(month)
```

```
## # A tibble: 336,776 x 19
## # Groups:   month [12]
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

`group_by()` doesn't change the data but, if you look closely at the output, you'll notice that it's now "grouped by" month. This means subsequent operations will now work "by month".

5.2 summarize()

The most important grouped operation is a summary. It collapses each group to a single row. Here we compute the average departure delay by month:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay)
  )
```

```
## # A tibble: 12 x 2
##    month delay
##    <int> <dbl>
##  1     1    NA
##  2     2    NA
##  3     3    NA
##  4     4    NA
##  5     5    NA
##  6     6    NA
##  7     7    NA
##  8     8    NA
##  9     9    NA
## 10    10    NA
## 11    11    NA
## 12    12    NA
```

Uhoh! Something has gone wrong and all of our results are `NA` (pronounced "N-A"), R's symbol for missing value. For now we'll remove them by using `na.rm = TRUE`:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE)
  )
```

```
## # A tibble: 12 x 2
##    month delay
##    <int> <dbl>
##  1     1 10.0
##  2     2 10.8
##  3     3 13.2
##  4     4 13.9
##  5     5 13.0
##  6     6 20.8
##  7     7 21.7
##  8     8 12.6
##  9     9  6.72
## 10    10  6.24
## 11    11  5.44
## 12    12 16.6
```

You can create any number of summaries in a single call to `summarize()`. One very useful summary is `n()`, which returns the number of rows in each group:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
```

```
## # A tibble: 12 x 3
##    month delay     n
##    <int> <dbl> <int>
##  1     1 10.0  27004
##  2     2 10.8  24951
##  3     3 13.2  28834
##  4     4 13.9  28330
##  5     5 13.0  28796
##  6     6 20.8  28243
##  7     7 21.7  29425
##  8     8 12.6  29327
##  9     9  6.72 27574
## 10    10  6.24 28889
## 11    11  5.44 27268
## 12    12 16.6  28135
```

5.3 Grouping by multiple variables

You can create groups using more than one variable. For example, we could make a group for each day:

```
daily <- flights |>
  group_by(year, month, day)
```

```
daily
```

```
## # A tibble: 336,776 x 19
## # Groups:   year, month, day [365]
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

When you summarize a tibble grouped by more than one variable, each summary removes the last group as a grouping variable.

```
daily_flights <- daily |>
  summarize(
    n = n()
  )
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the
## `.groups` argument.
```

```
daily_flights
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day     n
##    <int> <int> <int> <int>
##  1  2013     1     1   842
##  2  2013     1     2   943
##  3  2013     1     3   914
##  4  2013     1     4   915
##  5  2013     1     5   720
##  6  2013     1     6   832
##  7  2013     1     7   933
##  8  2013     1     8   899
##  9  2013     1     9   902
## 10  2013     1    10   932
## # ... with 355 more rows
## # i Use `print(n = ...)` to see more rows
```

If you're happy with this behavior, you can explicitly request it in order to suppress the message:

```
daily_flights <- daily |>
  summarize(
```

```
    n = n(),
    .groups = "drop_last"
  )
```

Alternatively, change the default behavior by setting a different value, e.g. `"drop"` to drop all grouping or `"keep"` to preserve the same groups.

5.4 Ungrouping

You might also want to remove grouping outside of `summarize()`. You can do this with `ungroup()`.

```
daily |>
  ungroup() |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    flights = n()
  )
```

```
## # A tibble: 1 x 2
##   delay flights
##   <dbl>   <int>
## 1  12.6  336776
```

As you can see, when you summarize an ungrouped data frame, you get a single row back because dplyr treats all the rows in an ungrouped data frame as belonging to one group.